Coursework 2 Report

Introduction

I have created a Pacman agent which is an optimal, utility-based agent, meaning it acts in a way which maximises all expected utilities. The agent works in a dynamic, accessible environment. The environment is also sequential as the direction in which Pacman moves affects what direction it will move in next. In addition to this, the environment is non-deterministic as there is only an 80% chance that Pacman will move in the direction which the agent wants to. It has a 20% chance of moving in a direction perpendicular to the intended direction.

Methodology

The first step in the method is creating a data structure which allows the agent to model the environment in such a way that it can easily work with the information given by the incoming precepts. The Grid class models the map as a 2-dimensional grid which is equal to the height and width of the layout used in the program. Each possible discrete location of the layout (including the walls) is assigned a coordinate value in the form (x, y). The value stored at (x, y) is utility of each coordinate. By default, each coordinate contains a utility of 0 and correct utilities are set later. This means I do not have to have the program to set walls to a utility value of 0, saving computation time.

The next step is to populate the grid representation of the map with the correct values. The utilities are calculated using the Bellamn equation. More precisely, I use the iterative formula so the values tend to the optimal policy for each step in the program. The iterative formula must be ran at every step due to the dynamic nature of the environment; the environment still changes even if Pacman does nothing. I only ran Bellmans update for the coordinates of empty spaces, capsules and food. This is because at those locations Pacman is in a non-terminal state so we must use sequential decision making as Pacman's direction of movement will affect later decisions. Hence it is important to take to the expected utilities of the coordinates around the four possible locations Pacman can move in to into consideration. It is unnecessary to run value iteration on the coordinates for walls as it is not possible for Pacman to move into those locations. In addition, it is unnecessary to run value iteration on the ghost locations as if Pacman moves into these locations then Pacman will be put in a terminal state (due to being eaten and therefore losing the game). This also helps keep the code efficient as less iterations of Bellmans update are required making the decision-making time of the agent quicker.

The reward function I have used is given by:

$$R(s) = \begin{cases} 0 \; if \; s = empty \; space \\ 3 \; if \; s = food \\ 3 \; if \; s = capsule \\ -100 \; if \; s = ghost \\ 50 \; if \; s = edible \; ghost \end{cases}$$

There is no reward for being in an empty space. There is a reward of 3 for Pacman to eat a piece of food or a capsule. There is a negative reward for ghosts as I want Pacman to not go where a ghost is. There is a positive reward for eating an edible ghost which is higher than the reward of eating food as we gain more points from eating a ghost than we do food hence we want to motivate the agent to prefer edible ghosts over food. I found these reward values balanced the game well so the agent performs as intended. To ensure the correct reward values were used in the computations, I checked if each possible coordinate was contained in the list of locations for different possible items. I firstly checked if they were a ghost that was not edible and then ghosts that were edible as these

are the smallest lists of locations. I then if the coordinate contained a capsule as this was the next smallest list. Next, the coordinate is checked to see if it contains a piece of food. Bellmans formula is ran with the correct reward values if the coordinate contains a food or capsule or edible ghost as these are non-terminal locations. Lastly, the coordinate is check to see if it a wall or not. This is the most efficient way I could find of finding which coordinates are empty. If a coordinate (i, j) does not contain a ghost, piece of food or capsule and it isn't a wall then it must be an empty space. Bellman is ran on the empty spaces with a reward value of 0.

Due to the non-deterministic nature of the environment, I have chosen to calculate the expected utility of moving in each direction. The agent uses these four values to decide which direction is best (which is the one with the highest expected utility). I use the following formula to find the maximum expected utility value:

$$\text{MEU} = \max[\ (0.8U_N + 0.1U_W + 0.1U_E), (0.8U_E + 0.1U_N + 0.1U_W), (0.8U_S + 0.1U_E + 0.1U_W), (0.8U_E + 0.1U_S + 0.1U_W)\ ]$$

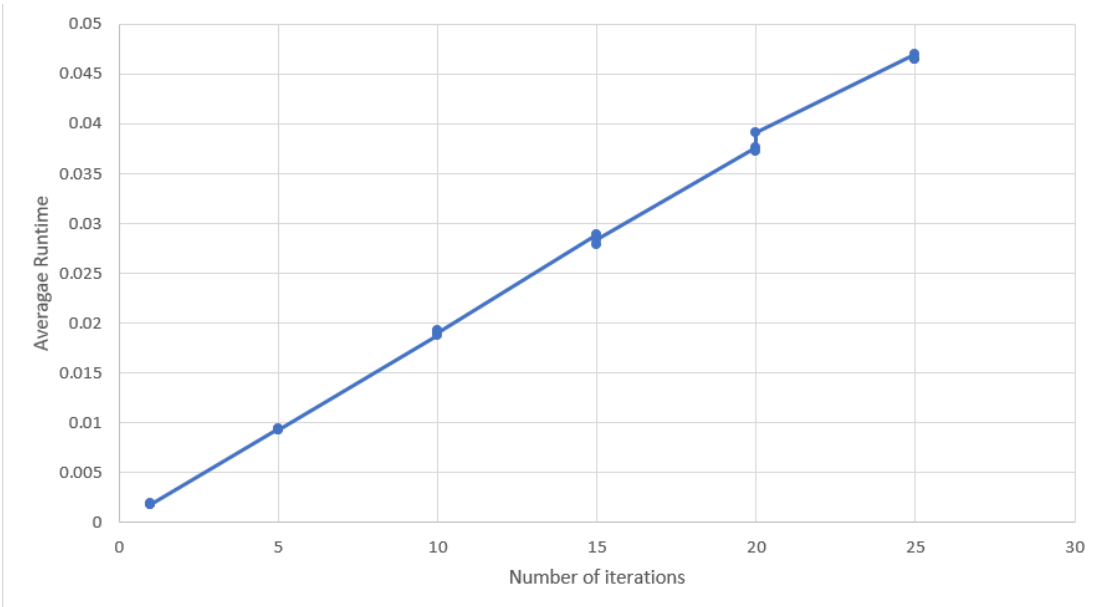
where U_d is the utility of the coordinate in direction d relative to Pacman's current position.

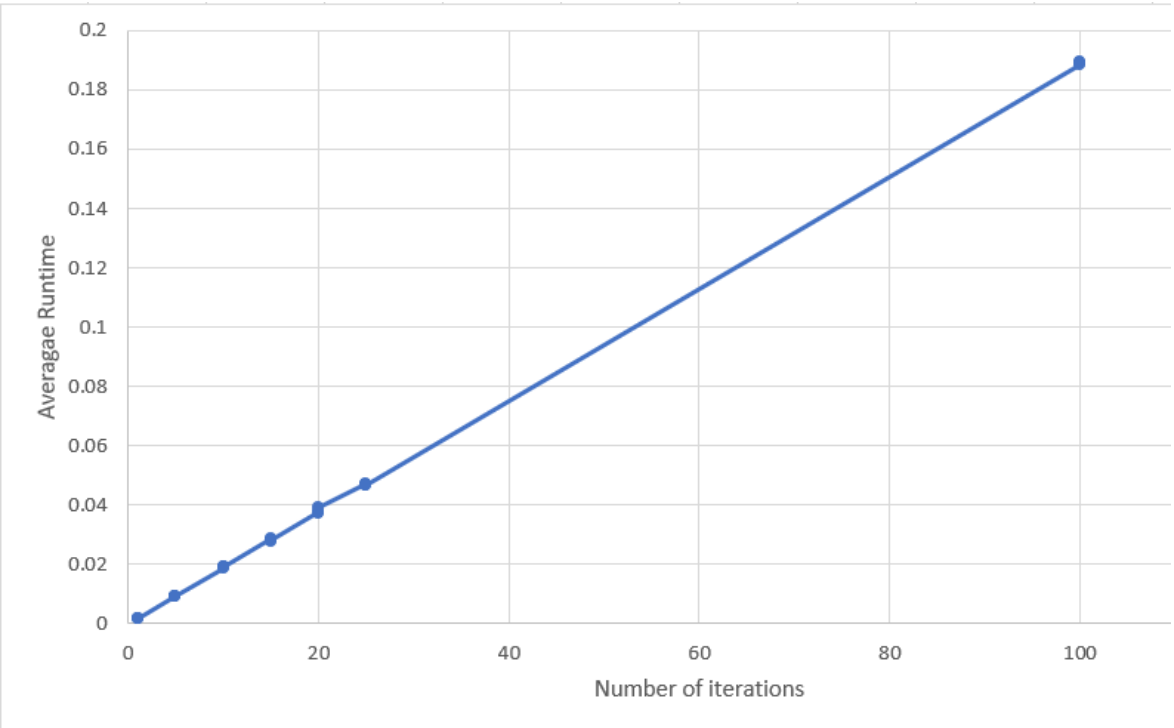
The intended direction which produce the highest maximum expected utility is the best direction for the Pacman to move.

To find an optimal number of iterations of Bellmans update I should run per step I carried out a series of tests. One of them was comparing the run time for different numbers of iterations. I test each number of iterations three times and plotted them on a scatter graph with a line of best fit to show the relationship between runtime and number of iterations. The runtime was recorded for each step and written to a CSV file where I used the data to create graphs in Microsoft Excel.

Iterations	Average 1	Average 2	Average 3
1	0.001831	0.001929	0.001835
5	0.009368	0.009313	0.009279
10	0.018826	0.019249	0.019035
15	0.028858	0.027937	0.028333
20	0.037576	0.037274	0.039176
25	0.047018	0.046462	0.046463
100	0.188393	0.189387	0.188987

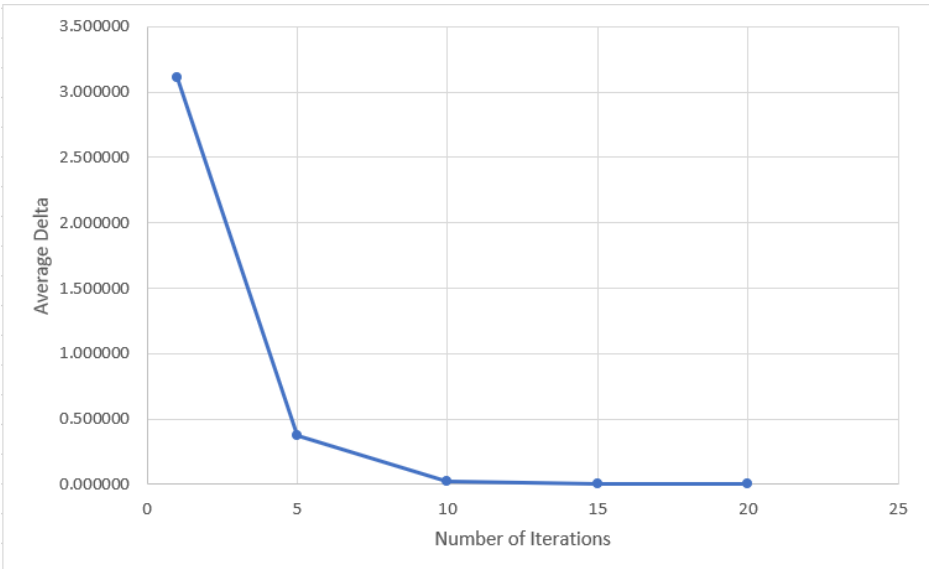
I decided to test 100 runs to interpolate a line of best fit between 100 iterations and 25. This would save time and I was already happy with assuming that there is a linear relationship between number of iterations and average run time based on my results for 1 to 25 iterations. My results show the average run time is proportional to number of iterations.





In my next test I assumed a number of iterations such as 100 was enough to find the optimal policy. The test I conducted was to see how many iterations it would take for the utilities to become close. I compared the final iteration value of the arbitrary point (2,5) for different numbers of iterations to the final iteration value of running Bellman update 100 times. Again, I used a CSV file to write the absolute delta values between my two test variables and Microsoft Excel to create graphs.

Iterations	Average Delta 1	Average Delta 2	Average Delta 3	**Average Delta**
1	2.098832	7.020531	0.221095	3.113486
5	0.692326	0.216789	0.225002	0.378039
10	0.002072	0.055225	0.000919	0.019405
15	0.005349	0.000171	0.002557	0.002692
20	0.000197	0.002038	0.001563	0.001266



The relationship is not linear here. After 10 iterations the delta value becomes very small. For this reason, I have chosen to run the Bellman update for 10 iterations due to the balance in time it takes to compute 10 iterations and the error it produces relative to the optimal policy values (assuming 100 iterations produces the optimal values).

Another way in which I tried to optimise computation times was using multiple threads to concurrently compute the Bellman update. The idea behind this was for one thread to compute Bellmans iterative equation for all coordinates and once it had reached the third row of the map, another thread would start from the first row. However, I had increased computation times of nearly double compared to iterating through the whole grid sequentially. After further researching the reason for this, I came to the conclusion that Python does not handle threads well enough to make it worthwhile using this approach.

Results

My results for the smallGrid layout with 1 ghost were the following:

```
Average Score: 66.53
Scores:        -503.0, -503.0, 496.0, -510.0, -503.0
502.0, 507.0, 489.0, -503.0, 505.0, -503.0, -510.0,
0, -516.0, 503.0, 501.0, -505.0, 501.0, 501.0, -514.
504.0, 503.0, 496.0, 498.0, -506.0, 503.0, 501.0, -5
Win Rate:      57/100 (0.57)
Record:        Loss, Loss, Win, Loss, Loss, Win, Los
 Loss, Loss, Win, Win, Loss, Loss, Win, Win, Win, Wi
 Loss, Loss, Loss, Win, Win, Win, Win, Win, Loss, Wi
```

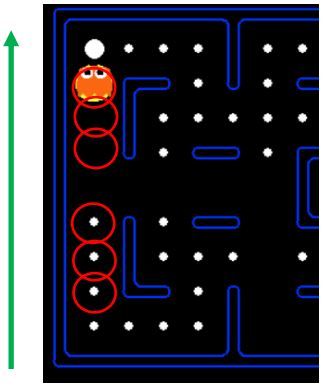
Due to the small size of the layout, I was able to quickly run 100 runs of my agent. I believe 100 is a good number of runs to minimise error. From the results I can see that my agent won the game 57% of the time.

My results for the mediumClassic layout with 2 ghosts were the following:

```
Average Score: 354.34
Scores:        878.0, -276.0, 1690.0, 1501.0, -
506.0, 1711.0, 26.0, 2088.0, -322.0, 800.0, -31
Win Rate:      10/50 (0.20)
Record:        Loss, Loss, Win, Win, Loss, Loss
oss, Loss, Loss, Loss, Loss, Loss, Loss, Loss,
```

Due to the larger size of the map compared to the smallGrid, I was only able to quickly run 50 tests. The results show my agent won 20% of the time.

From conducting tests where I watched the movement of my Pacman agent, I could see that the most common reason for the Pacman being eaten by the ghosts was when Pacman was being chased by the ghost. Due to the uncertainty in which direction Pacman would actually move in, if the agent instructed Pacman to move South (as the ghost is north of Pacman's position), then there is a 10% chance it instead moves West and a 10% chance that it instead moves East. If there was a wall east or west of Pacman then Pacman would stop therefore allowing itself to being eaten by the ghost that was one coordinate away. I will demonstrate an example of this below



At each point (shown by a red circle) there is a 20% chance that Pacman will stop as there is a 20% chance it'll move in the direction of a wall. As Pacman moves in the direction of the green allow, being directly followed a ghost, there is a 15.1% chance it'll successfully move north 7 times in a row and then east when it reaches the top left coordinate. This means if the ghost follows Pacman from (1,1) to (1,9) there is only a 15.1% chance that Pacman will survive which is low. If at any point Pacman moves into a wall, it will get eaten by ghost. This is due to the non-deterministic nature of the environment and therefore I am limited to how well I can improve this.

However, improvements can be made to the Bellman equation to take into account nearby negative utilities. The standard Bellman equation is:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} \Pr(s'|s,a)U(s')$$

Here we can see that the discount factor is multiplied by the maximum expected utility. This means the utility-based agent I have created is an optimal agent; it will maximise the sum expected utilities. In other words, it will always move to the coordinate with the highest expected utility value. This generally means negative or low positive utilities are not taken into account. Hence Pacman will only move away from a coordinate of negative utility value (for a non-edible ghost) if it is one space away from it. To take the negative values into consideration, it may be more useful to use the average instead of max which would allow the utility of the coordinate near a negative coordinate to also take a negative value, telling the agent that Pacman should not go there as it would be put next to a ghost. This could increase the probability of Pacman surviving the scenario given before as it would be more than one space away from the ghost. This would mean Pacman could stop once and still survive.